pyaardvark Documentation

Release 0.1

Kontron Europe GmbH

Jul 20, 2017

Contents

Contents:

CHAPTER 1

Introduction

The *pyaardvark* module tries to provide a very simple API to use the Total Phase Aardvark I²C/SPI Host adatper within your python program.

Simple Example

In this example we access an I²C-EEPROM on address *0x50* and read the first five bytes of its content:

```
import pyaardvark

a = pyaardvark.open()
data = a.i2c_master_write_read(0x50, '\x00', 5)
# data = '\x00\x01\x02\x03\x04'
a.close()
```

Easy, huh?

For those, who are not familiar with I²C-EEPROM accesses: You first write the offset to read from to the device (*0x00* in the example above) and then you read the desired amount of bytes from the device. The offset counter will automatically be incremened. Therefore, in the example above you read the bytes at the offsets 0, 1, 2, 3 and 4. Please note, that there are byte- and word-addressable EEPROMs. In this example we assumed a byte-addressable one, because our offset is only one byte.

Tutorial

Opening an Aardvark device

You have three choices to open your Aardvark device. The first is the one you saw in the simple example above:

```
a = pyaardvark.open()
```

If you have only one device connected to your machine, this is all you have to do. *pyaardvark.open()* automatically uses the first device it finds.

If you have multiple devices connected, you can either use the port parameter:

```
a = pyaardvark.open(1)
```

or the serial number, which you can find on the device itself or in your USB properties of your machine:

```
a = pyaardvark.open(serial_number='1111-222222')
```

In all cases *pyaardvark.open()* returns an *pyaardvark.Aardvark* object, which then can be used to access the host adapter.

Using the context manager protocol to open an Aardvark device

All methods of the *pyaardvark.Aardvark* object can raise an `IOError`. Instead of using *try .. except .. finally .. you can use the with* statement to open the device. Closing the device will then happen automatically after the block:

```
with pyaardvark.open() as a:
    print a.api_version
# no need for a.close() here
```

Accessing your I²C and SPI devices

To issue I²C or SPI transactions you have to first configure the adapter in the corresponding output mode. Each interface, I²C or SPI, can either be GPIOs or the actual interface. So if, for example you want to use both I²C and SPI at the same time and none of them as GPIOs:

```
a.enable_i2c = True
a.enable_spi = True
```

After you enabled the I²C interface you can issue transactions on the bus:

```
a.i2c_master_write(0x50, '\x00\x02\0x00\x00')
```

This will write adress device *0x50* and sends the byte sequence *0x00*, *0x02*, *0x00*, *0x00* to it. To read from a device use *pyaardvark.Aardvark.i2c_master_read()*. Eventually, both can be combined and issued in one transaction: *pyaardvark.Aardvark.i2c_master_write_read()*.

Closing the device

Releasing the device can be done with *pyaardvark.Aardvark.close()*:

```
a.close()
```

FAQ

Convert data to a string

Most parameters of the pyaardvark API take (byte) strings (eg. *pyaardvark.Aardvark.i2c_master_write_read()* etc). You can convert iterables to strings using the built-in chr function:

```
data = (0x01, 0xaf, 0xff)
data = ''.join(chr(c) for c in data) # data is '\x01\xaf\xff'
a.i2c_master_write(0x50, data)       # writes 1h, AFh, FFh to address 50h
```

To convert a character/string to a number you can use the build-in ord function:

```
data_str = a.i2c_master_read(0x50, 3) # data_str is '\xc0\x01\xff'
data = [ord(b) for b in data_str]     # data is [192, 1, 255]
```

CHAPTER 2

API

Module Interface

pyaardvark.**find_devices**()

Return a list of dictionaries. Each dictionary represents one device.

The dictionary contains the following keys: port, unique_id and in_use. *port* can be used with *open()*. *serial_number* is the serial number of the device (and can also be used with *open()*) and *in_use* indicates whether the device was opened before and can currently not be opened.

Note: There is no guarantee, that the returned information is still valid when you open the device. Esp. if you open a device by the port, the unique_id may change because you've just opened another device. Eg. it may be disconnected from the machine after you call *find_devices()* but before you call *open()*.

To open a device by its serial number, you should use the *open()* with the *serial_number* parameter.

---

pyaardvark.**open**(*port=None*, *serial_number=None*)

Open an aardvark device and return an *Aardvark* object. If the device cannot be opened an IOError is raised.

The *port* can be retrieved by *find_devices()*. Usually, the first device is 0, the second 1, etc.

If you are using only one device, you can therefore omit the parameter in which case 0 is used.

Another method to open a device is to use the serial number. You can either find the number on the device itself or in the in the corresponding USB property. The serial number is a string which looks like *NNNN-MMMMMMM*.

Raises an IOError if the port (or serial number) does not exist, is already connected or an incompatible device is found.

Note: There is a small chance that this function raises an IOError although the correct device is available and not opened. The open-by-serial-number method works by scanning the devices. But as explained in *find_devices()*, the returned information may be outdated. Therefore, *open()* checks the serial number

once the device is opened and if it is not the expected one, raises `IOError`. No retry mechanism is implemented.

As long as nobody comes along with a better idea, this failure case is up to the user.

Constants

Most API functions will throw an `IOError` in case an error is encountered. The `errno` attribute will set to one of the following values:

pyaardvark.**ERR_UNABLE_TO_LOAD_LIBRARY**

pyaardvark.**ERR_UNABLE_TO_LOAD_DRIVER**

pyaardvark.**ERR_UNABLE_TO_LOAD_FUNCTION**

pyaardvark.**ERR_INCOMPATIBLE_LIBRARY**

pyaardvark.**ERR_INCOMPATIBLE_DEVICE**

pyaardvark.**ERR_COMMUNICATION_ERROR**

pyaardvark.**ERR_UNABLE_TO_OPEN**

pyaardvark.**ERR_UNABLE_TO_CLOSE**

pyaardvark.**ERR_INVALID_HANDLE**

pyaardvark.**ERR_CONFIG_ERROR**

pyaardvark.**ERR_I2C_NOT_AVAILABLE**

pyaardvark.**ERR_I2C_NOT_ENABLED**

pyaardvark.**ERR_I2C_READ_ERROR**

pyaardvark.**ERR_I2C_WRITE_ERROR**

pyaardvark.**ERR_I2C_SLAVE_BAD_CONFIG**

pyaardvark.**ERR_I2C_SLAVE_READ_ERROR**

pyaardvark.**ERR_I2C_SLAVE_TIMEOUT**

pyaardvark.**ERR_I2C_DROPPED_EXCESS_BYTES**

pyaardvark.**ERR_I2C_BUS_ALREADY_FREE**

pyaardvark.**ERR_SPI_NOT_AVAILABLE**

pyaardvark.**ERR_SPI_NOT_ENABLED**

pyaardvark.**ERR_SPI_WRITE_ERROR**

pyaardvark.**ERR_SPI_SLAVE_READ_ERROR**

pyaardvark.**ERR_SPI_SLAVE_TIMEOUT**

pyaardvark.**ERR_SPI_DROPPED_EXCESS_BYTES**

The functions *Aardvark.i2c_slave_read()*, *Aardvark.i2c_master_read()* and *Aardvark.i2c_master_write()* will throw an `IOError` with its `errno` attribute set to a status code to indicate the state of the I2C transaction in case of an error.

pyaardvark.**I2C_STATUS_OK**
No error occured.

pyaardvark.**I2C_STATUS_BUS_ERROR**
A bus error has occured. Transaction was aborted.

pyaardvark.**I2C_STATUS_SLA_ACK**
Bus arbitration was lost during master transation; another master on the bus has successfully addressed this Aardvark adapter's slave address. As a result, this Aardvark adapter has automatically switched to slave mode and is responding.

pyaardvark.**I2C_STATUS_SLA_NACK**
The Aardvark adapter failed to receive acknowledgement for the requested slave address during a master operation.

pyaardvark.**I2C_STATUS_DATA_NACK**
The last data byte in the transaction was not acknowledged by the slave.

pyaardvark.**I2C_STATUS_ARB_LOST**
I2C master arbitration lost, because another master on the bus was accessing the bus simultaneously.

pyaardvark.**I2C_STATUS_BUS_LOCKED**
An I2C packet is in progress and the time since the last I2C event executed or received on the bus has exceeded the bus lock timeout. This is most likely due to the clock line of the bus being held low by some other device or due to the data line held low such that a start condition cannot be executed by the Aardvark adapter. The bus lock timeout can be configured using the *Aardvark.i2c_bus_timeout* property. The Aardvark adapter resets its own I2C interface when a timeout is observed and no further action is taken on the bus.

pyaardvark.**I2C_STATUS_LAST_DATA_ACK**
The last byte was ACK'ed by the opposing master. When the Aardvark slave is configured with a fixed length transmit buffer, it will detach itself from the I2C bus after the buffer is fully transmitted and the aardvark slave also expects that the last byte sent from this buffer is NACK'ed by the opposing master device.

Aardvark Object

class pyaardvark.**Aardvark**(*port=0*)
Represents an Aardvark device.

api_version = None
Version of underlying C module (aardvark.so, aardvark.dll) as a string. See *hardware_revision* for more information on the format.

close()
Close the device.

disable_i2c_monitor()
Disable the I2C monitor.

Raises an IOError if the hardware adapter does not support monitor mode.

disable_i2c_slave()
Disable I2C slave mode.

enable_i2c
Set this to *True* to enable the hardware I2C interface. If set to *False* the hardware interface will be disabled and its pins (SDA and SCL) can be used as GPIOs.

enable_i2c_monitor()
Activate the I2C monitor.

Enabling the monitor will disable all other functions of the adapter.

Raises an `IOError` if the hardware adapter does not support monitor mode.

enable_i2c_slave(*slave_address*)
Enable I2C slave mode.

The device will respond to the specified slave_address if it is addressed.

You can wait for the data with *poll()* and get it with *i2c_slave_read*.

enable_spi
Set this to *True* to enable the hardware SPI interface. If set to *False* the hardware interface will be disabled and its pins (MISO, MOSI, SCK and SS) can be used as GPIOs.

firmware_version = None
Firmware version of the host adapter as a string. See *hardware_revision* for more information on the format.

handle = None
A handle which is used as the first paramter for all calls to the underlying API.

hardware_revision = None
Hardware revision of the host adapter as a string. The format is `M.NN` where *M* is the major number and *NN* the zero padded minor number.

i2c_bitrate
I2C bitrate in kHz. Not every bitrate is supported by the host adapter. Therefore, the actual bitrate may be less than the value which is set.

The power-on default value is 100 kHz.

i2c_bus_timeout
I2C bus lock timeout in ms.

Minimum value is 10 ms and the maximum value is 450 ms. Not every value can be set and will be rounded to the next possible number. You can read back the property to get the actual value.

The power-on default value is 200 ms.

i2c_master_read(*addr*, *length*, *flags=0*)
Make an I2C read access.

The given I2C device is addressed and clock cycles for *length* bytes are generated. A short read will occur if the device generates an early NAK.

The transaction is finished with an I2C stop condition unless the I2C_NO_STOP flag is set.

i2c_master_write(*i2c_address*, *data*, *flags=0*)
Make an I2C write access.

The given I2C device is addressed and data given as a string is written. The transaction is finished with an I2C stop condition unless I2C_NO_STOP is set in the flags.

10 bit addresses are supported if the I2C_10_BIT_ADDR flag is set.

i2c_master_write_read(*i2c_address*, *data*, *length*)
Make an I2C write/read access.

First an I2C write access is issued. No stop condition will be generated. Instead the read access begins with a repeated start.

This method is useful for accessing most addressable I2C devices like EEPROMs, port expander, etc.

Basically, this is just a convenient function which internally uses *i2c_master_write* and *i2c_master_read*.

i2c_monitor_read()
Retrieved any data fetched by the monitor.

This function has an integrated timeout mechanism. You should use *poll()* to determine if there is any data available.

Returns a list of data bytes and special symbols. There are three special symbols: *I2C_MONITOR_NACK*, I2C_MONITOR_START and I2C_MONITOR_STOP.

i2c_pullups
Setting this to *True* will enable the I2C pullup resistors. If set to *False* the pullup resistors will be disabled.

Raises an `IOError` if the hardware adapter does not support pullup resistors.

i2c_slave_last_transmit_size
Returns the number of bytes transmitted by the slave.

i2c_slave_read()
Read the bytes from an I2C slave reception.

The bytes are returned as a string object.

i2c_slave_response
Response to next read command.

An array of bytes that will be transmitted to the I2C master with the next read operation.

Warning: Due to the fact that the Aardvark API does not provide a means to read out this value, it is buffered when setting the property. Reading the property therefore might not return what is actually stored in the device.

poll(*timeout=None*)
Wait for an event to occur.

If *timeout* is given, if specifies the length of time in milliseconds which the function will wait for events before returing. If *timeout* is omitted, negative or None, the call will block until there is an event.

Returns a list of events. In case no event is pending, an empty list is returned.

spi_bitrate
SPI bitrate in kHz. Not every bitrate is supported by the host adapter. Therefore, the actual bitrate may be less than the value which is set. The slowest bitrate supported is 125kHz. Any smaller value will be rounded up to 125kHz.

The power-on default value is 1000 kHz.

spi_configure(*polarity*, *phase*, *bitorder*)
Configure the SPI interface.

spi_configure_mode(*spi_mode*)
Configure the SPI interface by the well known SPI modes.

spi_ss_polarity(*polarity*)
Change the ouput polarity on the SS line.

Please note, that this only affects the master functions.

spi_write(*data*)
Write a stream of bytes to a SPI device.

target_power
Setting this to *True* will activate the power pins (4 and 6). If set to *False* the power will be deactivated.

Raises an `IOError` if the hardware adapter does not support the switchable power pins.

unique_id()
Return the unique identifier of the device. The identifier is the serial number you can find on the adapter without the dash. Eg. the serial number 0012-345678 would be 12345678.

unique_id_str()
Return the unique identifier. But unlike *unique_id()*, the ID is returned as a string which has the format NNNN-MMMMMMM.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

O

P

S

T

U